AFRL-VA-WP-TM-2006-3009

COMPUTATIONAL FLUID DYNAMICS (CFD) MODELING AND ANALYSIS

Delivery Order 0006: Cache-Aware Air Vehicles Unstructured Solver (AVUS)

MGNet
8 South Street
Cos Cob, CT 06807-1618

Dr. Craig C. Douglas

Yale University

Adam F. Zornes

University of Kentucky

AUGUST 2005

Final Report for 01 January 2005 – 31 August 2005

STINFO FINAL REPORT

AIR VEHICLES DIRECTORATE
AIR FORCE MATERIEL COMMAND
AIR FORCE RESEARCH LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Wright Site (AFRL/WS) Public Affairs Office (PAO) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

PAO Case Number: AFRL/WS-06-0092, 10 Jan 2006.

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.

/s/

VICTOR S. BURNLEY
Project Engineer

/s/

REID MELVILLE, Chief
Computational Sciences Branch

/s/

DOUGLAS C. BLAKE, Chief
Aeronautical Sciences Division
Air Vehicles Directorate

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
August 2005	Final	01/01/2005– 08/31/2005

4. TITLE AND SUBTITLE

COMPUTATIONAL FLUID DYNAMICS (CFD) MODELING AND ANALYSIS
Delivery Order 0006: Cache-Aware Air Vehicles Unstructured Solver (AVUS)

5a. CONTRACT NUMBER
F33615-03-D-3307-0006

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0602201

6. AUTHOR(S)

Dr. Craig C. Douglas (Yale University)
Adam F. Zornes (University of Kentucky)

5d. PROJECT NUMBER
A00G

5e. TASK NUMBER

5f. WORK UNIT NUMBER
0D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MGNet
8 South Street
Cos Cob, CT 06807-1618

Yale University

University of Kentucky

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Vehicles Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7542

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL-VA-WP

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-VA-WP-TM-2006-3009

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

PAO Case Number: AFRL/WS-06-0092, 10 Jan 2006.

14. ABSTRACT

The Air Vehicles Unstructured Solver (AVUS) is a three-dimensional (3-D) flow simulator running on unstructured grids. Embedded in the simulator is a sparse matrix package for solving coupled partial differential equations. Unstructured grid solvers usually do not exhibit optimal computer memory access patterns. The purpose of this report is to detail the memory access patterns and provide a roadmap for improving it.

15. SUBJECT TERMS

CFD, cache memory, computational performance, unstructured meshes

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Victor S. Burnley
Unclassified	Unclassified	Unclassified	SAR	16	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255- 4305

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

1 Task Objective

The objective of this delivery order is to improve the cache performance of the computer code AVUS.

1.1 Background

AVUS is a three-dimensional (3-D) flow simulator running on unstructured grids. Embedded in the simulator is a sparse matrix package for solving coupled partial differential equations. Unstructured grid solvers usually do not exhibit optimal computer memory access patterns. This project will determine the memory access patterns and provide a roadmap for improving it.

2 Description of Work

The following summarizes the activity / work accomplished during the period of January 1 to August 31, 2005 by the principal investigator Dr. Craig C. Douglas and his graduate student Adam Zornes.

A tutorial, *Cache-aware Algorithms and PDE Libraries*, on using computer cache memory subsystems was presented by Craig Douglas on May 18, 2005 at Wright-Patterson Air Force Base to the Computational Sciences Branch of the Air Vehicles Directorate. The presentation covers how the caches are organized and what is required to use caches effectively.

Years ago, knowledge of the number of floating point multiplies used in a given algorithm (or code) provided a good measure of the running time of a program. This was a good measure for comparing different algorithms to solve the same problem. This is no longer true. Many current computer designs, including the node architecture of most parallel supercomputers, employ caches and hierarchical memory architecture. Therefore, the speed of a code (e.g., multigrid) depends increasingly on how well the cache structure is exploited. The number of cache misses provides a better measure for comparing algorithms than the number of multiplies. Unfortunately, estimating cache misses is difficult to model a priori and only somewhat easier to do a posteriori.

Typical multigrid applications are running on data sets that are much too large to fit into the caches. Thus, copies of the data that are once brought into the cache should be reused as often as possible. For multigrid, the possible number of reuses is always at least as great as the number of iterations of the smoother (or rougher), plus the residual correction before correction steps. Suitable fixed and adaptive blocking strategies for both structured and unstructured grids must be introduced. Both types of cache-aware algorithms use a fixed, given percentage of the cache.

Fixed algorithms use fixed blocks of the unknowns (or subdomains) that are determined in a preprocessing step. Adaptive algorithms use a moving active set of unknowns that should be in cache and can be reused. Once an unknown has been completely updated for k iterations, it leaves the active set.

The cache-aware algorithms improve the cache usage without changing the underlying algorithm. In particular, bitwise compatibility is guaranteed between the standard and the high-performance implementations of the algorithms. This is illustrated by comparisons for various multigrid algorithms on a selection of different computers for problems in two and three dimensions.

To apply cache-aware principles to the code AVUS, it is first necessary to read the actual code, find the relevant parts, and start profiling running examples once the code has been built on suitable platforms. The techniques we intend to introduce should be fairly portable. One aspect that we had not expected is that AVUS is built using a pair of scripts that are tied to certain commercial compilers on specific platforms. We do not

have access to these compilers and have had to produce variants of the scripts just to get started on compiling and running AVUS.

We have successfully built and run AVUS on two platforms: a Xeon 64-EMT-based PC cluster and an Itanium^2-based shared-memory HP Integrity Superdome. The code was profiled to find the routines that need to be considered for cache-aware algorithms.

The compilation is driven by the script make_avus. One must configure options for the platform on which compilation is to be done. The script then builds and links executables for the desired platform and options. The main option is single or double precision. Double precision in the default. The make_avus script first checks to see if the ParMetis library has been built for the desired platform. If not, it calls another script, make_metis, to build the desired library. After the library has been established, the make_avus script checks to see if the .f sources have been built for the desired platform and are up to date. If not, they are built and saved in the objects directory en mass. When the .f sources have been built, a few other sources are built and then all are linked together with the ParMetis library to create the executable, which is saved in the bin directory.

The compilation scripts must be modified to allow for profiling. This involves adding the correct options for compilation (specific to each compiler) for producing profiling output. This must be done both in the make_avus and make_metis scripts, and sources must be rebuilt accordingly.

Execution is generally done through a .job file. This file sets several environment variables (including executable location and calling sequence), creates and input file for the program, and calls the program execution script. The execution script is AvusRUN. This first moves various files to the correct locations as set by the .job file. The script then runs the executable and tidies up by moving output files to a location set by the .job file and destroying temporary files.

To correctly collect profiling data, some aspects of the execution process must be modified. This may include altering the AvusRUN script to not remove temporary files and may also entail some other changes to the script.

The first step in profiling was to run AVUS under *gprof*, a relatively simple profiler. We then had statistics on how much time was spent in each of the routines in the overall code. This is helpful in determining where the code spends most of its time on a single processor. A complete *gprof* output for one of examples starts at about 70 pages of text, which we omit and will provide to the program manager electronically.

The top 15 routines are shown in Table 1.

Table 1: Primary Routines Based on *gprof* Statistics

% Time	Cumsecs	Seconds	Calls	Msec/Call	Name
14.7	0.25	30.24	6	5040	*Ucm6*
12.2	55.28	25.04	5	5008	*Karl6sc*
9.7	75.28	20.00	1	20000	*Poorgrd*
8.4	92.56	17.28	1	17280	*Orderpt*
6.6	106.08	13.52	5	2704	*Dfdqi6sc*
6.2	118.72	12.64	5	2528	*Lhslusc*
5.9	130.80	12.08	5	2416	*Dfdqv6sc*
4.7	140.48	9.68	6	1613	*Riemann*
3.2	146.96	6.48	40	162	*Hunter*
3.0	153.04	6.08	5	1216	*Dfdqt6sc*
2.3	157.76	4.72	6	787	*Slip*
2.0	161.92	4.16	6	693	*Grad6*
2.0	166.00	4.08	5	816	*Vflux6*
1.7	169.52	3.52	5	704	*Preset6sc*
1.4	172.32	2.80	5	560	*Posi*

Ucm6 constructs left and right face values of primitives. *Poorgrd* compute a mask array for the one-sided least-squares gradient construction at each face. the mask array is ‘good’ and it is initialized to true in subroutine zero. A neighbor cell not truly one-sided will cause a false entry in the appropriate location in the array. *Orderpt* removes redundant points from each face to ensure that each edge is composed of two points and is shared by two faces of any given cell in three dimensions. It also orders points around the perimeter of each face.

Karl6sc is the parallel symmetric Gauss-Seidel iterative solver. It communicates twice per iteration with neighbors. *Riemann* solves a Riemann problem at every face using the exact Riemann solver of Gottlieb and Groth.

Most of the rest of the routines are used to construct Jacobians by computing fluxes or enforcing boundary conditions or positivity requirements. Anything that passes through data once only is not a candidate for improving cache reuse unless the routines can be combined so that fewer passes are made through the same data. In particular, *Poorgrd*, *Orderpt*, and *Hunter* are not candidates for improvement since on very large data sets they use relatively little time.

A more sophisticated profiler is the HPCToolkit developed at Rice University. It has the capability to profile multiple processor codes on platforms that are supported by the Performance Application Programming Interface (PAPI), which was developed at the University of Tennessee at Knoxville. We are in the process of building scripts that will allow us to extract highly detailed runtime statistics on the performance of AVUS. Unfortunately, writing the scripts is time consuming, tedious, and not automated. When completed, the scripts will be useful after the project as AVUS evolves over time.

In the case of multiple processors, the grid is automatically reordered to take advantage of locality. This is not done automatically in the single processor case. However, a program called Blacksmith may be run to reorder the grid before starting AVUS. This causes a significant drop in the time spent per iteration.
AVUS is spending most of its time generating the problems it has to solve, not on solving these problems. One of the principal iterative solvers, a block symmetric Gauss-Seidel, performs two major communications exchanges every iteration. This algorithm needs to be modified in order to be cache efficient. One possibility is to eliminate some of the communication by switching the algorithm to a more traditional Schwarz alternating method that runs multiple steps of Gauss-Seidel locally.

Schwarz procedures solve local (domain decomposed) sub-problems to some level of accuracy. There are two variants of interest:

1. Additive: all sub-problems are solved in parallel. Data is exchanged between sub-problems for overlapping data (if any) in parallel, too.
2. Multiplicative: each sub-problem is solved sequentially. Data is exchanged only between sub-problems that have an update pending.

The additive form is most suitable for parallel processing. There are multilevel variants as well, e.g., multigrid within each sub-problem versus domain decomposition within a global multigrid procedure. I prefer the latter, but the domain decomposition community prefers the former.

The advantage of an additive Schwarz procedure for AVUS is that the symmetric Gauss-Seidel procedure can be modified to be cache-aware. Currently, there are two parallel communications of sub-domain boundary data per iteration. Several complete iterations can be done instead before communication. Switching to a natural order Gauss-Seidel will make it even easier to add cache-aware algorithms devised either by my group or the one of Ulrich Ruede in Erlangen (both our groups have cooperated on a number of joint papers). Many papers can be found in the MGNet preprint web page.

Keeping the right-hand sides and the matrix values together in memory in one structure seems to make a modest improvement in cache locality. For 3-D problems, this is not always the case, however. In a large application, it may be better to have the right-hand side in two places depending on what it is needed for. Copying the data makes sense if there will be many iterations of the Gauss-Seidel solver.

Another approach is to do a domain decomposition style Gauss-Seidel solver within each Schwarz sub-problem, where the domains now are localized to the size of the usable part of cache. Jonathan Hu wrote a Ph.D. dissertation on how to do this for multigrid on a single processor for unstructured grids. Danny Thorne wrote a Ph.D. dissertation on how to do this for multigrid on a single processor for structured, adaptively refined grids. Both dissertations are available in the ML-DDDAS web page, http://www.mgnet.org/~douglas/ml-dddas.html. Codes can be provided as examples of how to develop such codes. Both students' codes fall in the category of research codes. A newer, production level code that works well in both environments is slowly being

written by the researchers in their spare time.

A significant idea in Hu's dissertation is to study the adjacency matrix C=(cij) of cache-bound subdomains. These matrices are square and the size of the number of cache blocks used to solve a problem. If cij= cji>0, then cache blocks *i* and *j* transfer data between each other. Trying to minimize the number of nonzeroes will reduce computational time. If the cij represent how many pieces of data have to be transferred from block *i* to *j*, minimizing the cij as well will have an impact on cache reusability.

Studying the techniques employed by either Hu or Thorne will take a significant amount of time.

3 Profiling AVUS with the HPCToolkit

To simplify the process of profiling AVUS with the HPCToolkit, the .job files and the AvusRun script have been combined with other HPCToolkit commands and options in a single script, hpc_script.

At the beginning of the script, there are a number of variables that must be defined. These are listed and discussed Table 2. Below this, the input file is created and the AvusRun script is concatenated. Alterations have been made, and as of now there is support for Linux platforms only. Support for other platforms can be added as needed.

The process for analyzing AVUS with the HPCToolkit involves five steps. The first step is to determine the loop structure of the binary executable file using bloop. Next, the executable is run under hpcrun with various optional events defined in the variable HPCRUN_EVENTS. After this has been done, the output files produced by hpcrun are analyzed by hpcprof. These files are then all combined by hpcquick into a database viewable by either hpcviewer or a web browser. Finally, the results may be viewed.

A number of variables may be defined in the script. Table 2 lists those variables and a short description of each. For proper execution, all path names should be fully qualified, not relative.

Table 2: HPC_SCRIPT Variable Definitions

Variable name	Description
AVUSLOC	the directory of the AVUS executable
SCRATCH	the directory to use for execution of the AVUS script
MPI_EXEC	the command to invoke a program under MPI, including any options
JOBLOC	the directory from where the script should be run
INPUT	the directory and name of the AVUS input file
OUTPUT	the directory and name of the AVUS output file to be created
PRECISION	the precision to use for AVUS, *single* or *double*
GRID	the directory and name of the AVUS grid file
INTERSECTION	the directory and name of the AVUS intersection file
RESTART	the directory and name of the AVUS restart file
NEW_RESTART	the directory and name of the AVUS restart file to create
PIX	the directory and name of the AVUS pix file

TAP_LOCATIONS	the directory and name of the AVUS tap locations file
SHUTDOWN	the directory and name of the AVUS shutdown file to be created
MOVIE_TAP	the directory and name of the AVUS movie tap file to create
B1_TRIP	the directory and name of the B1 trip file to create
BOUNDARY_CONDITIONS	the directory and name of the boundary conditions file
OVERWRITE	the overwrite flag, *overwrite* or *nooverwrite*
OS_TYPE	the operating system (OS) used; currently only *Linux* is supported

Table 3 contains definitions of the relevant variable used by the HPCToolkit. See the web site http://www.hypersoft.rice.edu for further information about the HPCToolkit. There are many options which are described in detail in various manual pages for the individual components of the HPCToolkit. The site contains tutorials for an earlier and largely incompatible version of the software.

Table 3: HPCToolkit Variable Definitions

Variable name	Description
BLOOP_FILE_NAME	the name for the bloop output file to be created
BLOOP_OPTIONS	options for running bloop
EXE	the directory and name of the executable to be run
EXE_ARGS	arguments used by the executable
HPCPROF_OPTIONS	options for running hpcprof
HPCRUN_EVENTS	events to be taken note of by hpcrun (a complete list of events that can be profiled can be obtained by executing the command *hpcrun –L*)
HPCRUN_OPTIONS	options for running hpcrun
HPCRUN_OUTPUT_FILE	prefix for output from hpcrun
OUTPUT_DIR	the directory to place output
SRC_DIRS	the directories where the source code for the executable may be found
Z_OR_X	determines the format of output from hpcquick, XML or HTML using *-x* or *-z* respectively

4 Sources of Future Cooperation

The program manager has computer access to our work through accounts on www.mgnet.org (a Mac Mini running Mac OS X 10.4) and hpc1c.csr.uky.edu (a Centos 4.1 based Itanium^2 HP workstation). HPCToolkit is installed and functional on hpc1c. Should the program manager need to run simple examples in parallel in our environment, we will issue logins on hpc1d and hpc1e. I expect that the program manager will have questions about how to deal with Schwarz alternating procedures and some of the sophisticated cache-aware algorithms that Jonathan Hu and Danny Thorne devised as part of their Ph.D. dissertations. I expect to continue cooperating with the program manager after the formal end of the project on an informal basis, including answering questions related to this final report.